Parallel and Distributed Systems - Graph Minor Report

Spyridon Baltsas - AEM: 10443

1 Summary

This report is about my proposed solution for the Graph Minor problem. To be specific, here will be briefly presented the algorithm that I have implemented for my approach, an estimate for its time complexity, its general efficiency in handling various datasets. In addition, we may see how it behaves when we use several Parallelization libraries and different number of threads each time. The source code, building instructions and how to pass arguments to the compiled binary can be found *in this repository* and its *README*.

2 The algorithm

The algorithm was designed with the following points in mind:

- Memory efficiency, since we may have to process very large datasets of sparse matrices with it
- Time complexity, to make the most out the cycles of our CPU as possible
- Parallelization. Split the data with the most efficient way possible, and keep the synchronization (mutexes/semaphores) at a minimum, without any data races of course

2.1 Single Thread

The program takes as inputs 2 files, the first one contains the sparse matrix of the graph in matrix market (MM) format, and the second one is the vector c, which has information about which node belongs to which cluster, thus it has a length equal to the nodes of the graph. Equivalently, it has length equal to the amount of the non zero values in the adjacency matrix on the graph, in our case the sparse matrix in MM format. It stores the input matrix in COO format, but the Graph Minor's adjacency matrix in a format quite similar to CSC (in code this is an array of `struct row`), which contains the following data:

1. The non zero elements of the row (`int non_zeros`)
2. The columns which contain the non zero elements of the row as an array (`int *columns`)
3. An array of the non zero values of the row (`int *values`).

Let A the input sparse matrix of graph and M the sparse matrix of the desired graph minor. The program calculates the matrix minor without the need for matrix Ω of the specification, thanks to the following observation:

- The value of the i-th non zero element of A input matrix will be added up to the Graph Minor adjacency matrix element in row equal to c[rowsA[i]-1]-1 and in column c[colsA[i]-1]-1. Or, in other words:

$$M[c[rowsA[i]-1]-1, c[colsA[i]-1]-1] += valuesA[i] \tag{2.1}$$

The (-1)s are needed because we assume the matrix starts from 0 and not 1.

2.2 Multi-threaded

The multi-threaded version of the algorithm use the same observation as above, but with some changes, in order to get parallelized effectively, without the need of synchronization using mutexes and signals. Therefore, we load each thread with a data structure (in code this is an array of `struct ThreadData`) containing the following information:

1. the ID of the thread (`int threadID`)
2. The amount of matrix A's non zeros whose thread is responsible for (`int non_zeros`)
3. An array of indexes of the matrix A's non zeros (`int *non_zeros_indexes`)

We split the non zero elements of matrix A (COO format) to the responsible threads according to which row of the graph minor is element's destination. Each thread is responsible only for a distinct range of rows in the resulting graph minor matrix. Let N the number of available threads. Again, we assume that matrix starts from 0. Thus, for this job we use a modulo operation as seen in source code:

$$c[rowsA[i] - 1] - 1 \mod N \tag{2.2}$$

To be more clear, this is the part of the code which is responsible for this:

```
for(int i = 0; i < MAX_THREADS; ++i){
      threadData[i].threadID = i;
      threadData[i].non_zeros_indexes = malloc(NON_ZEROS_PER_THREAD[i]*sizeof(int));
      threadData[i].non_zeros = NON_ZEROS_PER_THREAD[i];
      counter = 0;
      for(int j = 0; j < NON_ZEROS; ++j){
          //split data among threads according to which row they belong in cluster adjacency matrix
          if((c[rowsA[j] -1 ] -1) % MAX_THREADS == i){
              threadData[i].non_zeros_indexes[counter] = j;
              counter++;
          }
      }
  }
```

This way, each thread can process its data independently from each other, without any race conditions. This of course, adds a small overhead.

2.3 Time complexity

Let n be the number of non zero elements of the input matrix, and m the number of non zero elements in the graph minor matrix. An approximation of the complexity of sequential algorithm would be $\mathcal{O}(m \cdot n)$. In case of multi-threaded, would be $\mathcal{O}\left(\frac{m \cdot n}{N}\right)$ where N is the number of available threads.

3 Test Specifications

Please take note that on the following processing times the duration of the overhead was omitted. The following operations were considered as "overhead", as it may be seen from source code as well:

- I/O Operations. In our case, reading the data from the matrix market and vector files and printing to `stdout` the resulting sparse matrix.
- Initialization and loading the required data structures before actually processing the input data.

3.1 Technical Specifications

The computer that was used to produce the following result times had the following specifications, processing power wise:

- Intel Core i5-8300H @ 2.30 GHz (4 cores, 8 threads)
- 8GB DDR4 RAM @ 2667 MHz

Finally, the operating system was Debian Linux.

3.2 Test Dataset Specifications

matrix name	rows	columns	non zero values
ca2010.mtx	710,145	710,145	3,489,366
pa2010.mtx	421,545	421,545	2,058,462
ri2010.mtx	25,181	25,181	125,750
tx2010.mtx	914,231	914,231	4,456,272
nj2010.mtx	169,588	169,588	829,912
road usa.mtx	23,947,347	23,947,347	57,708,624

4 Results

4.1 Single Thread

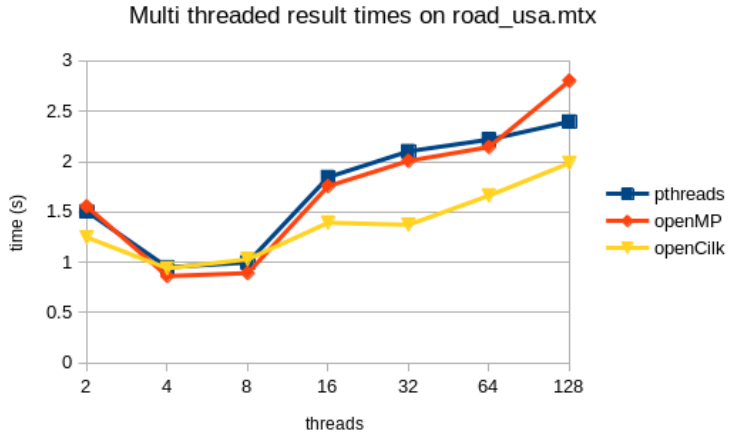
Datasets	Sequential time (seconds)	Matlab time (seconds)
road usa.mtx	2.429773	4.80690
ca2010.mtx	0.146844	0.12822
pa2010.mtx	0.081838	0.07003
ri2010.mtx	0.006248	0.00402
tx2010.mtx	0.181871	0.17025
nj2010.mtx	0.032843	0.02731

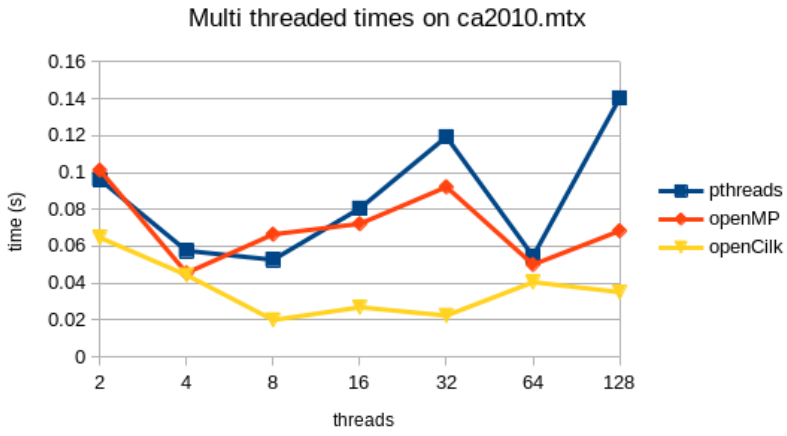
4.2 Multi-threaded

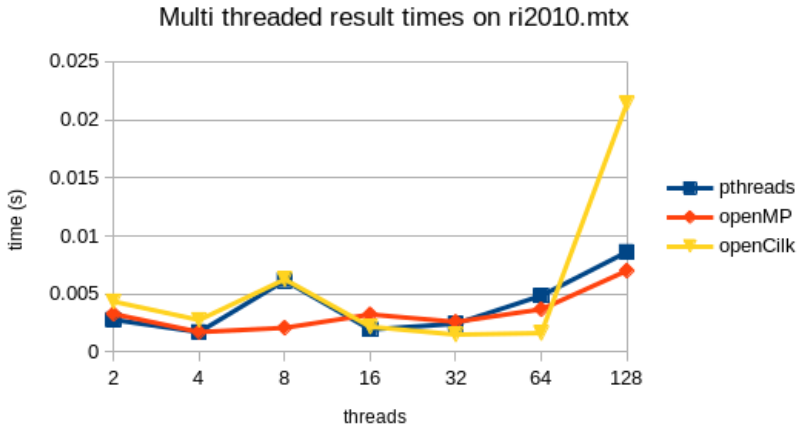
The parallelization libraries that were used are according to the specification, P-threads, OpenMP and OpenCilk. The tests used 2,4,8,16,32,64 and 128 threads each time. It was noticed that above 128 there was no point of testing, because the result times were either similar to these ones of 128 threads or even worse, because the threads were too many for the processor to handle. In addition, a significant speedup is noticed when threads are between 4 and 8 , or even 4 and 16 in some cases, depending on the dataset. Last but not least it appears that OpenCilk is more efficient than the other two when we increase the number of threads and OpenMP tends to be a little more efficient than pthreads.

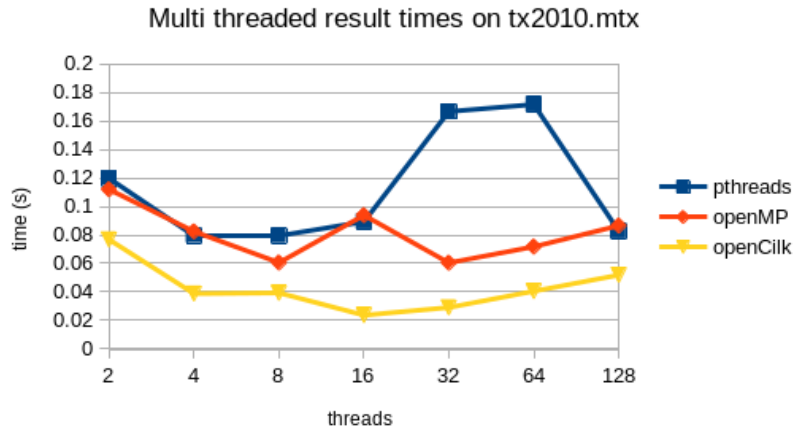
4.2.1 Charts

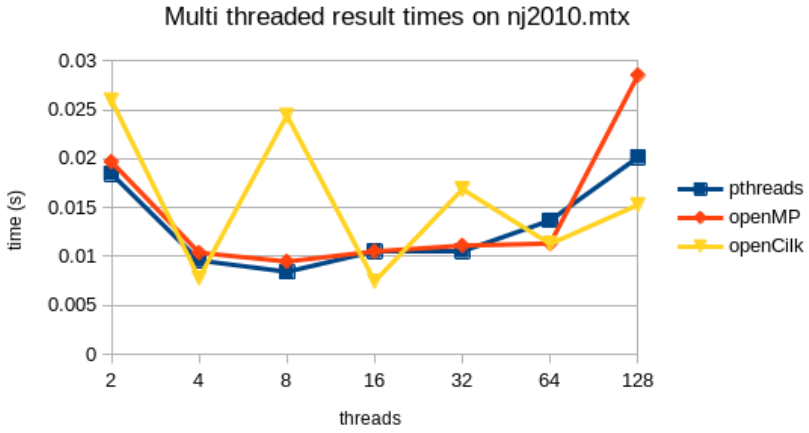
Here are the result times of the parallel algorithm using the required libraries for various number of threads, summed up into charts.






Multi threaded result times on ri2010.mtx
time (s)
0.025
0.02
0.015
0.01
0.005
0
2
4
8
16
32
64
128
threads
pthreads
openMP
openCilk


Multi threaded result times on tx2010.mtx
time (s)
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
2
4
8
16
32
64
128
threads
pthreads
openMP
openCilk


Multi threaded result times on nj2010.mtx
time (s)
0.03
0.025
0.02
0.015
0.01
0.005
0
2
4
8
16
32
64
128
threads
pthreads
openMP
openCilk